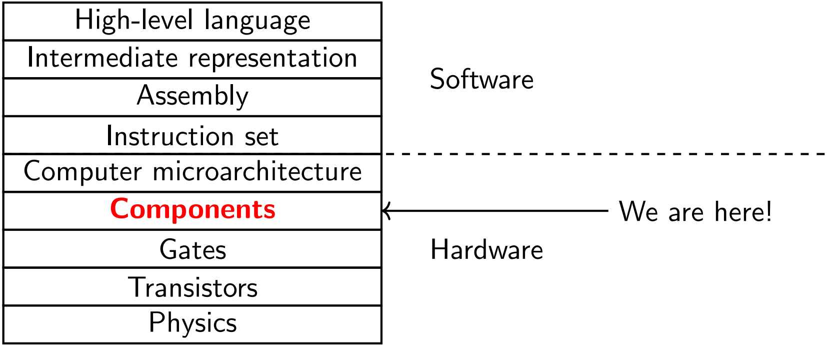
Building with flip-flops and registers
COMSM1302 Overview of Computer Architecture

John Lapinskas, University of Bristol

From gates to components



Things are getting complex enough that we can't keep thinking of individual NAND gates any more!

We must abstract things into components/subcircuits — plexers, adders, registers, gates, and NANDs *only* when they are the right tool for the job.

These components will themselves become part of larger components...

Registers as a building block: A counter

A **counter** stores and outputs a binary value *out* that goes up by 1 at (the rising edge of) each clock cycle. It has a single input: *reset*. If *reset* is 1 on a rising edge, the value in the counter is set back to 0.

How can we build a 16-bit counter from a 16-bit register?

Registers as a building block: A counter

A **counter** stores and outputs a binary value *out* that goes up by 1 at (the rising edge of) each clock cycle. It has a single input: *reset*. If *reset* is 1 on a rising edge, the value in the counter is set back to 0.

How can we build a 16-bit counter from a 16-bit register?

Let's take the same approach as with the register and draw an informal "truth table" to express our next desired output out_{new} in terms of our inputs and the output out_{old} from the previous cycle.

reset	out_{new}
0	$out_{\mathrm{old}} + 1$
1	0x0000

Registers as a building block: A counter

A **counter** stores and outputs a binary value *out* that goes up by 1 at (the rising edge of) each clock cycle. It has a single input: *reset*. If *reset* is 1 on a rising edge, the value in the counter is set back to 0.

How can we build a 16-bit counter from a 16-bit register?

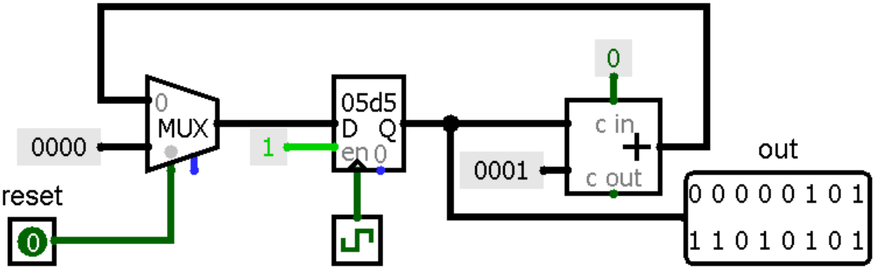
Let's take the same approach as with the register and draw an informal "truth table" to express our next desired output out_{new} in terms of our inputs and the output out_{old} from the previous cycle.

reset	out_{new}
0	$out_{\mathrm{old}} + 1$
1	0x0000

We already know how to calculate $out_{\mathrm{old}} + 1$ from out_{old}: with an adder!

And as with the register, we can express this sort of "if zero then x otherwise y" logic neatly with a multiplexer.

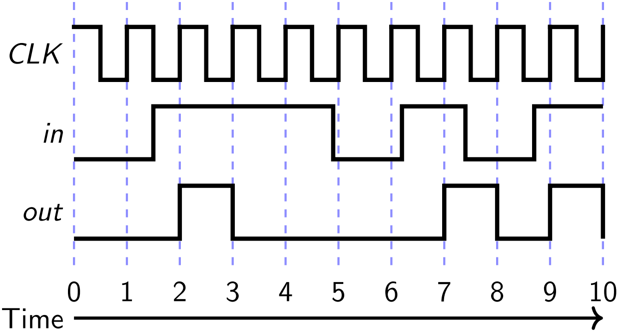
The counter in Logisim



[See video for a demonstration and further explanation.
The circuit is available for download from the unit page.]

Flip-flops as a building block: A one-shot

A **one-shot** turns a 1 input into a pulse of 1 lasting for a single clock cycle:



This is useful for e.g. initialising registers at power-on, or turning long button-presses from a user into a more convenient form.

How can we build this out of flip-flops? It helps to break the circuit's behaviour down into **states** stored in flip-flops/registers.

State diagrams

We want the one-shot to:

a. Wait for a 1 on *in*.
b. Send a pulse for one cycle.
c. Wait for a 0 on *in*, then go back to (a).

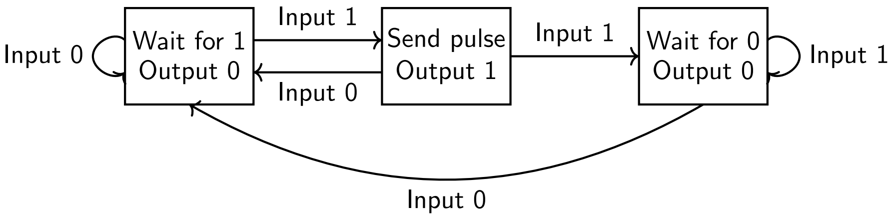
We can represent this as a diagram:

State diagrams

We want the one-shot to:

a. Wait for a 1 on *in*.
b. Send a pulse for one cycle.
c. Wait for a 0 on *in*, then go back to (a).

We can represent this as a diagram:

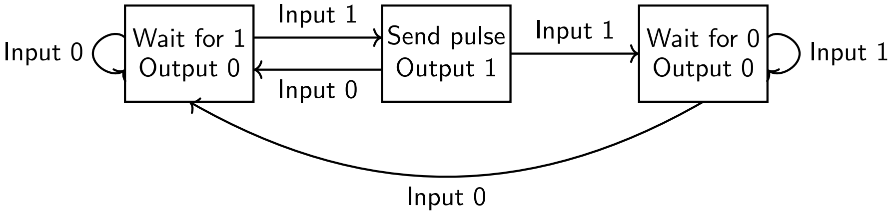


State diagrams

We want the one-shot to:

a. Wait for a 1 on *in*.
b. Send a pulse for one cycle.
c. Wait for a 0 on *in*, then go back to (a).

We can redraw this more compactly:

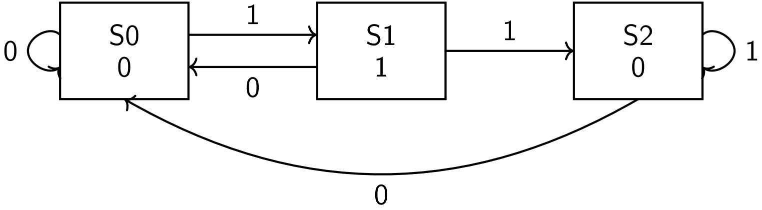


State diagrams

We want the one-shot to:

a. Wait for a 1 on *in*.
b. Send a pulse for one cycle.
c. Wait for a 0 on *in*, then go back to (a).

We can redraw this more compactly:

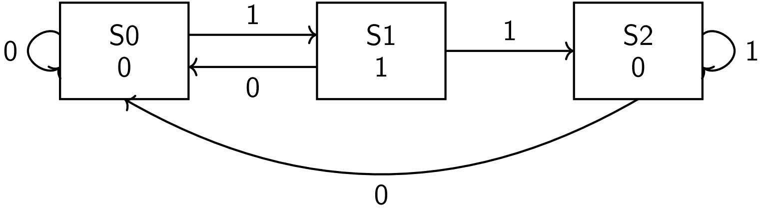


State diagrams

We want the one-shot to:

a. Wait for a 1 on *in*.
b. Send a pulse for one cycle.
c. Wait for a 0 on *in*, then go back to (a).

We can redraw this more compactly:



This is a **Moore machine**. It transitions between states each clock cycle based on its input, and its output is a function of its state.

We can store the state S as (e.g.) a binary number in a register/flip-flops. Then state transitions and outputs come from combinatorial logic!

Building the one-shot

Say we store the state in two flip-flops XY, representing S0, S1 and S2 as 00, 01 and 10 respectively. Then our truth tables are:

X_{old}	Y_{old}	in	X_{new}	Y_{new}
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	0
1	1	X	X	X

We can take

$$X_{\text{new}} = (X_{\text{old}} \vee Y_{\text{old}}) \wedge in,$$

$$Y_{\text{new}} = \neg X_{\text{old}} \wedge \neg Y_{\text{old}} \wedge in.$$

Building the one-shot

Say we store the state in two flip-flops XY, representing S0, S1 and S2 as 00, 01 and 10 respectively. Then our truth tables are:

X_{old}	Y_{old}	*in*	X_{new}	Y_{new}
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	0
1	1	X	X	X

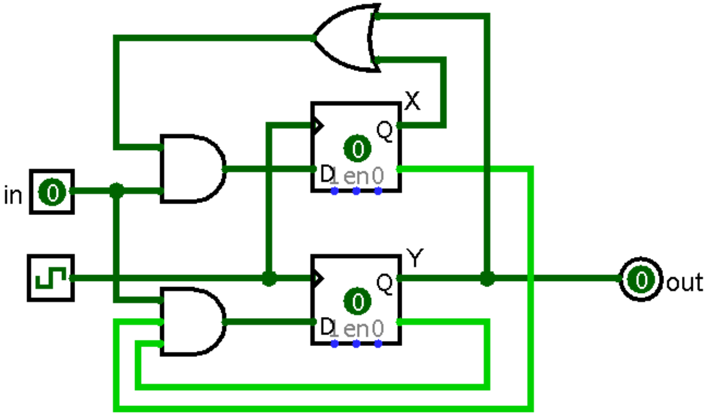
We can take

$$X_{\mathrm{new}} = (X_{\mathrm{old}} \vee Y_{\mathrm{old}}) \wedge in,$$
$$Y_{\mathrm{new}} = \neg X_{\mathrm{old}} \wedge \neg Y_{\mathrm{old}} \wedge in.$$

X	Y	*out*
0	0	0
0	1	1
1	0	0
1	1	X

We can take $out = Y$.

The one-shot in Logisim



[See video for a demonstration and further explanation.
The circuit is available for download from the unit page.]

A better one-shot: Mealy machines

Is this optimal? Definitely not! There are many tricks to do better, e.g. being very careful about encoding your state in the most useful way.

Here's one trick: By capturing the input in its own flip-flop/register, we can make the output depend on both the old state and the old input.

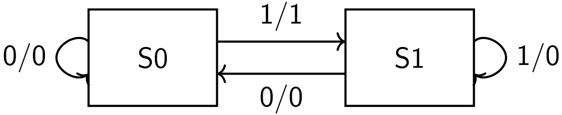
This means we can make the output depend on **transitions** between states, not just the state itself.

A better one-shot: Mealy machines

Is this optimal? Definitely not! There are many tricks to do better, e.g. being very careful about encoding your state in the most useful way.

Here's one trick: By capturing the input in its own flip-flop/register, we can make the output depend on both the old state and the old input.

This means we can make the output depend on **transitions** between states, not just the state itself.



Here e.g. "1/0" written near an arrow means that the transition between states happens on input 1, and results in output 0.

This is called a **Mealy machine**. Both Moore and Mealy machines are examples of **finite state machines**.

Building a better one-shot

The truth table we need to make this Mealy machine is:

S_{old}	in_{old}	S_{new}	out
0	0	0	0
0	1	1	1
1	0	0	0
1	1	1	0

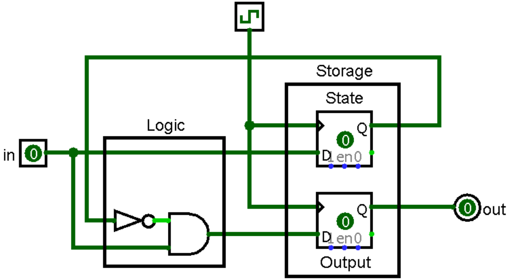
$$S_{\mathrm{new}} = in_{\mathrm{old}},$$
$$out = \neg S_{\mathrm{old}} \wedge in_{\mathrm{old}}.$$

Building a better one-shot

The truth table we need to make this Mealy machine is:

S_{old}	in_{old}	S_{new}	out
0	0	0	0
0	1	1	1
1	0	0	0
1	1	1	0

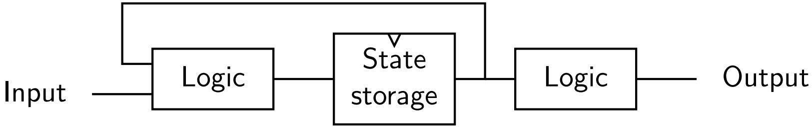
$$S_{\text{new}} = in_{\text{old}},$$
$$out = \neg S_{\text{old}} \wedge in_{\text{old}}.$$



Much better! Notice we could also remove the NOT gate if we wanted, by using the Q' output of the state flip-flop instead of the Q output.

General Moore and Mealy machines

You can implement **any** Moore/Mealy machine with the following schema:



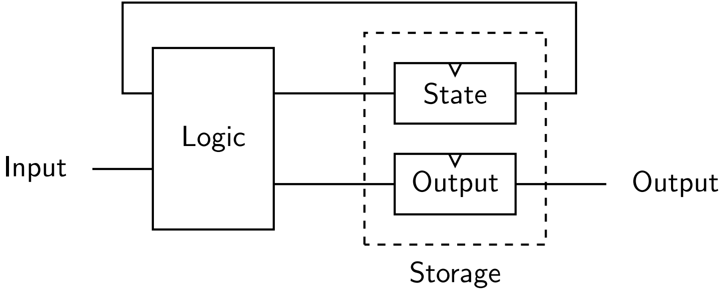
Every Moore machine

The storage can be flip-flops or registers, but should be clocked.

The logic should be combinatorial (i.e. unclocked gates), and you can implement it by writing down the truth tables — like with the one-shot.

General Moore and Mealy machines

You can implement **any** Moore/Mealy machine with the following schema:



Every Mealy machine

The storage can be flip-flops or registers, but should be clocked.

The logic should be combinatorial (i.e. unclocked gates), and you can implement it by writing down the truth tables — like with the one-shot.

Less formal state diagrams

Suppose we want a vending machine to:

a. Wait for the user to enter the first digit of their snack.
b. Wait for the user to enter the second digit of their snack or hit "back" to erase the first digit.
c. Wait for the user to put in enough money to pay for their snack or hit "back" to cancel the transaction.
d. Dispense the snack (if the user didn't hit "back") and any change, then go back to (a).

This won't be a "pure" Moore or Mealy machine — e.g. we'll want to track the amount of money via a register. But we can and should still build the circuit by tracking an internal state.

Less formal state diagrams

Suppose we want a vending machine to:

a. Wait for the user to enter the first digit of their snack.
b. Wait for the user to enter the second digit of their snack or hit "back" to erase the first digit.
c. Wait for the user to put in enough money to pay for their snack or hit "back" to cancel the transaction.
d. Dispense the snack (if the user didn't hit "back") and any change, then go back to (a).

This won't be a "pure" Moore or Mealy machine — e.g. we'll want to track the amount of money via a register. But we can and should still build the circuit by tracking an internal state.

Finite state machines are useful even in programming, as a way of simplifying complex control logic. For example, the platforming logic of Celeste is available here, and is based around a finite state machine!